Evaluating the Effectiveness of a Vector-Length-Agnostic Instruction Set

Andrei Poenaru[1] and Simon McIntosh-Smith[1]

Department of Computer Science
University of Bristol
Bristol, United Kingdom
cssnmis@bristol.ac.uk

Abstract In this paper we evaluate the efficacy of the Arm Scalable Vector Extension (SVE) instruction set for HPC workloads using a set of established mini-apps. Exploiting the vector capabilities of SVE will be a key factor in achieving high performance on upcoming generations of Arm-based processors. SVE is a flexible instruction set, but its design is fundamentally different from other contemporary SIMD extensions, such as AVX or NEON, which could present a challenge to its adoption. We use a selection of mini-apps which covers a wide range of scientific application classes to investigate SVE, using a combination of static and dynamic analysis. We inspect how SVE capabilities are used in the mini-apps' kernels, as generated by all SVE compilers available at the time of writing, for both arithmetic and memory operations. We compare our findings against similar data gathered on currently available processors. Although the extent to which vector code is generated varies by mini-app, all compilers tested successfully utilise SVE to vectorise *more* code than they are able to when targeting NEON, Arm's previous-generation SIMD instruction set. For most mini-apps, we expect performance improvements as SVE width is increased.

Keywords: Instruction sets · SVE · Vectorisation · SIMD · Data parallelism

1 Introduction

Modern processors rely on SIMD hardware to provide high performance for scientific applications. Vector hardware is not a new concept, with its origins reaching back to the CRAY-1 in 1975, but taking advantage of such capabilities has become increasingly important over the past few years.

Current x86-based processors offer SIMD capabilities through the 256-bit AVX2 and 512-bit AVX-512 instruction sets. Arm-based alternatives, however, have so far only offered 128-bit vectors through the instruction set previously known as NEON, which is now part of the ARMv8 Advanced SIMD (ASIMD) instruction group. The relatively short width of ASIMD vectors, combined with the reduced flexibility of this instruction set originally designed for media and signal processing, has limited the performance of Arm-based processors on a number of scientific applications [1].

The next generations of high-performance Arm processors will use the Scalable Vector Extension (SVE) to provide more powerful vector operations [2]. Unlike current SIMD implementations, SVE is a vector-length-agnostic (VLA) instruction set, allowing each implementation to choose a vector width between 128 and 2048 bits, in increments of 128 bits, with SVE binaries being portable between implementations. The first SVE-capable hardware will become available in 2020 [3], but a number of tools that enable SVE experiments through either emulation or simulation are already available. In this paper, we use these SVE tools to assess the efficacy of the new vector instruction set across a range of common HPC problem classes.

This paper makes several contributions:

- A comparison of the vectorisation efficiency of several HPC mini-apps on contemporary vector platforms from Arm and Intel;
- An analysis of SVE usage on the mini-apps, inspecting executed vector code and memory access patterns and their relation to SVE vector widths;
- An evaluation of the state of currently available SVE compilers and performance analysis tools.

2 Background

Initially vital in many-core devices such as GPUs [4] and the Intel Xeon Phi [5], vector code is now important in *all* high-performance processors. Utilising the wide vector units in the latest generations of x86 processors is the only way to approach peak performance [6].

Vector code is generally produced by optimising compilers, but compiler-backed auto-vectorisation cannot be assumed to be optimal [7, 8]. Therefore, it is important to evaluate its effectiveness on new hardware platforms. Furthermore, differences in instruction sets and their implementation in hardware can cause different behaviour on two distinct processors, even when the same benchmark and toolchain are used.

On x86 processors there are many variants of AVX available, and the optimal code for each variant may be significantly different [9], but with the Arm SVE instruction set, the generated machine code does not depend on a fixed vector width. Instead, executables automatically exploit the widest vector size available at run-time, using an approach similar to that of the very first vector computers [10]. This is particularly attractive for benchmarks based on real-world scientific applications, as they tend to steer clear of platform- or vendor-specific optimisations and instead opt for portable code.

Mini-apps are benchmarks built by reducing full-size scientific applications to the smallest implementation that preserves its performance characteristics, while eliminating non-critical features such as input/output [11]. The main computation kernels in mini-apps are closely similar — sometimes identical — to those in their parent applications. Mini-apps are also lightweight in terms of dependencies, not requiring specialised libraries to run, which often contrasts with large-scale scientific applications. The mini-apps used in this paper have previously been used as part of a comprehensive benchmarking suite for studying

the performance of a new supercomputer, Isambard, the first production-ready system based on ARMv8 processors [1, 12]; McIntosh-Smith et al. also present an extensive overview of the current status of Arm in HPC.

SVE will be implemented in upcoming generations of Arm-based HPC processors, including the Fujitsu A64FX [13] and the Marvell ThunderX4 [14]. Because SVE supports vector widths between 128 and 2048 bits, chip designers need to select the vector width to be used in their implementation. It is, thus, important to estimate how this choice will affect the performance of applications run on such future processors, and experiments are already being run to determine the impact of SVE width on scientific kernels [15].

3 Methodology

In this paper, we study the efficacy of SVE over a number of mini-apps, each representative of a different class of scientific problems. The applications use only OpenMP or MPI, require no external libraries, and rely on automatic vectorisation by the compiler, i. e. no platform-specific intrinsics are used. The mini-apps studied are: STREAM, the established memory bandwidth benchmark [16]; BUDE, a molecular docking application developed at the University of Bristol [17]; TeaLeaf, a heat-diffusion mini-app [18]; CloverLeaf, a hydrodynamics code that solves Euler's equations of compressible fluid dynamics [19]; MegaSweep, a STREAM-style benchmark that uses the main kernel from SNAP, a deterministic discrete ordinates transport proxy application [20]; Neutral, a Monte-Carlo neutral particle transport mini-app [21]; and MiniFMM, a Fast Multipole Method mini-app that uses OpenMP tasks for parallelisation [22].

We performed the experiments described in this paper using a combination of static and dynamic analysis tools. The compilers used were the latest versions of the three main SVE toolchains available at the time of writing: Arm HPC Compiler 19.2, GCC 8.2, and Cray Compiler (CCE) 9.0; for SVE, a pre-release version of the Cray Compiler, 9.0a, was used. We enabled most compiler optimisation with the flags `-O3 -ffast-math -mcpu=thunderx2t99+sve`; full reproducibility details can be found in Section 8. In all experiments, we used a single OpenMP thread and MPI process (where applicable), and the inputs were chosen such that the non-instrumented run time is below 5 seconds on a single core of a ThunderX2 processor. We used compiler optimisation listings and annotated source code to count vectorised loops in each mini-app, and we confirmed that vector instructions are run using hardware counters.

Because no SVE-equipped hardware is available today, we ran the SVE versions of the mini-apps using the Arm Instruction Emulator (ArmIE)[1]. ArmIE runs base AArch64 instructions natively on the host, and switches to emulation when encountering SVE instructions. It also allows user-defined instrumentation code, known as instrumentation *clients*, to be run over both the native and emulated parts of the application. We used custom instrumentation clients to record data about the instructions executed and the memory accesses performed

[1] `https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-instruction-emulator`

by the programs. We limited instrumentation to the core computation kernels in the mini-apps, such that data is not collected for the initialisation and shutdown stages of the applications, because these are generally not important when measuring real-world performance. Recording data outside the kernels can skew the results by showing a misleadingly high number of scalar instructions if these sections are not optimised for vectorisation. To define the regions where data was collected, we inserted special instructions to start and stop instrumentation, which are invalid AArch64 instructions but are recognised and honoured by our ArmIE client.

We classified dynamically recorded instructions into several categories: scalar AArch64 (A64), vector AArch64 (i.e. Advanced SIMD/NEON), SVE arithmetic, SVE memory loads, SVE memory stores, SVE moves, and SVE control flow. We used the memory access trace data to describe each operation as ⟨load/store, contiguous/non-contiguous, some/all vector lanes active⟩. The SVE vector width was set by stepping through the powers of two between 128 and 2048.

4 Results

4.1 Compiler Vectorisation Efficiency

We analysed the static vectorisation efficiency of SVE compared to AVX by looking at certain loops in the kernels of each mini-app. We selected loops to cover the majority of the mini-apps' run times, as reported by a profiled run on a real ThunderX2 processor. For targeting Arm, both with SVE and NEON, we used the three main HPC compilers: Arm's HPC compiler, GCC, and the Cray Compiler; for x86, we used the same versions of GCC and Cray, but we used the Intel Compiler 19.0 instead of the Arm HPC Compiler.

Table 1 shows, for each application, the number of loops considered, the percentage of run time that they represent, and the number of loops vectorised by each compiler on each platform. We show TeaLeaf twice — once using a CG solver, once using a PPCG solver — because the two runs cover very different code paths, and both are representative of real workloads. There are no MiniFMM results with the Cray Compiler because the application's build system does not currently support the Cray Compiler.

Aggregating the results across mini-apps, we observed that the compilers which can generate code for all the instructions sets vectorised the highest number of loops on SVE.

We then studied the factors influencing vectorisation on each mini-app individually. TeaLeaf with the PPCG solver was fully vectorised on all the platforms, by all compilers. TeaLeaf with CG and BUDE achieved 80% or more vectorisation with all compilers; it should be possible to achieve full vectorisation, as shown by the Intel compiler on AVX and GCC on Arm. CloverLeaf and MiniFMM showed all loops except one vectorised with Arm, Cray, and Intel, but only about half with GCC; GCC reports that further vectorisation is not beneficial according to its cost model, on all platforms, due to indirect access. MegaSweep was not vectorised by GCC on any platform, but fully vectorised by Cray on SVE and Intel on x86, which suggests vectorisation is possible, but not

Table 1. Number of loops vectorised by each compiler on the top loop-nests, selected by percentage of total run time on a ThunderX2 processor, in the mini-apps studied. The results for AVX2 and AVX-512 were identical; here they share the *AVX* label.

Application	% Time (Total Loops)	SVE			NEON			AVX		
		Arm	Cray	GCC	Arm	Cray	GCC	Intel	Cray	GCC
STREAM	92.4 (4)	4	4	4	4	4	4	4	4	4
BUDE	98.6 (4)	4	3	3	3	4	3	4	4	3
TeaLeaf (cg)	87.2 (8)	5	6	8	5	6	8	8	6	6
TeaLeaf (ppcg)	91.2 (6)	6	6	6	6	6	6	6	6	6
CloverLeaf	62.5 (10)	9	10	6	8	9	6	10	9	8
MegaSweep	70.3 (4)	1	4	0	1	1	0	4	1	0
Neutral	85.8 (2)	0	0	0	0	0	0	0	0	0
MiniFMM	98.1 (8)	7	—	5	3	—	5	7	—	5
Total	(46)	36	32	32	30	30	32	43	28	32

all compilers understand the loops' structure. Neutral was not vectorised at all, on any platform, due to the deeply nested branching in its algorithm.

When targeting x86, all compilers vectorised the same number of loops on both AVX2, e. g. for Broadwell, and AVX-512, e. g. for Skylake.

4.2 Dynamic Instruction Analysis

After we obtained vectorised code for the mini-apps, we recorded dynamic instruction execution traces at each power-of-two SVE vector length between 128 and 2048 bits. We added a NEON-only and a non-vectorised (scalar) run for each application, to serve as baselines against which to compare the SVE results. The traces allowed us to identify the types of SVE instructions executed and how their dynamic count varies with the chosen vector length.

Figure 1 shows the dynamic instruction count analysis for the **STREAM** benchmark, where instructions are grouped by type: scalar AArch64, NEON (AArch64 ASIMD), and several groups of SVE operations; a lower number of instructions executed is generally better. In the scalar and NEON-only cases, the Arm and Cray Compiler showed similar behaviour, but the GCC version ran more than twice as many instructions because it did not make use of load/store pair instructions, an operation in which two 64-bit values can be read from/written to memory in a single instruction. When targeting SVE, all three compilers performed similarly, and we saw a decrease in the total instruction count as we increased vector length, since each instruction had increasingly more active lanes. No compiler generated load/store pairs for SVE, so the instruction count at 128 bits — the same vector length that NEON uses — is close to that observed for GCC when targeting NEON. The Arm and Cray compilers, but not GCC, chose to use scalar A64 instructions for loop control flow, which resulted in the the scalar instruction count also varying with SVE width.

BUDE, a heavily compute-bound application, ran vector code almost exclusively, which results in a clear inverse relation between the dynamic instruction

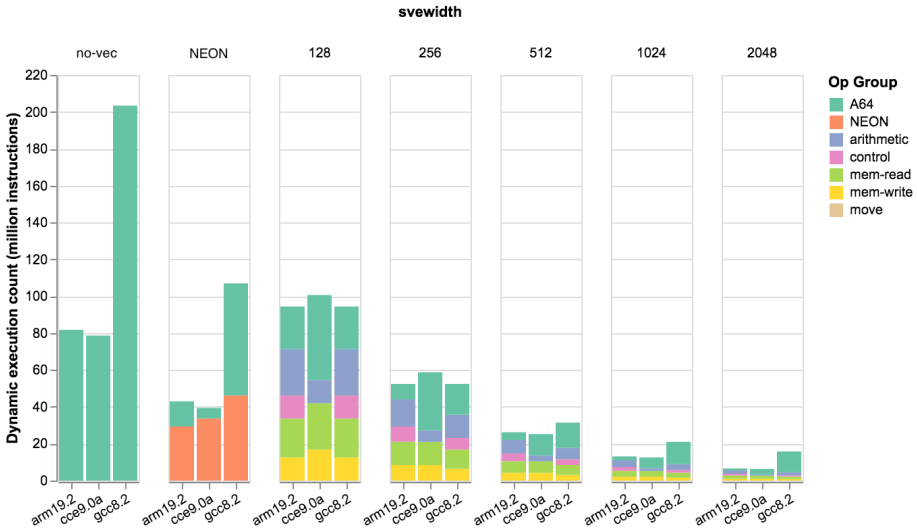

Figure 1. Dynamic instruction count and grouping for STREAM. Lower is generally better. *A64* refers to scalar instructions; *NEON* refers to base-AArch64 ASIMD vector instructions; the remaining groups are all SVE instructions.

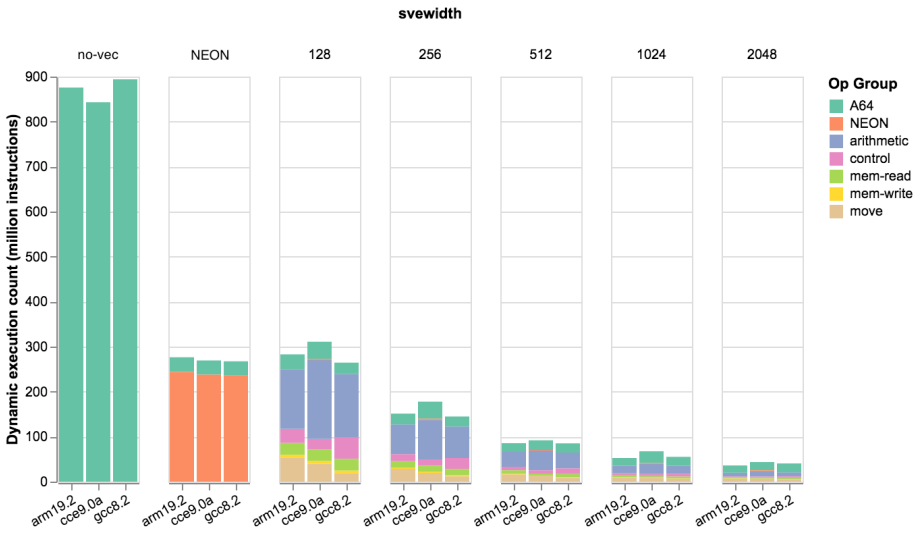

Figure 2. Dynamic instruction count and grouping for BUDE. Lower is generally better. *A64* refers to scalar instructions; *NEON* refers to base-AArch64 ASIMD vector instructions; the remaining groups are all SVE instructions.

count and the vector length. All compilers performed very similarly for this application. The results are shown in Figure 2.

TeaLeaf and **CloverLeaf** exhibited similar behaviour: the code was only partially vectorised, leading to a mixture of SVE and scalar instructions. As the SVE length was increased, the number of executed SVE instructions decreased, but the number of scalar instructions executed stayed constant. The non-SVE part comes largely from outer-loop code, since in these cases only the innermost loop is vectorised by the compilers. Figure 3 shows the CloverLeaf results; TeaLeaf follows an almost-identical profile.

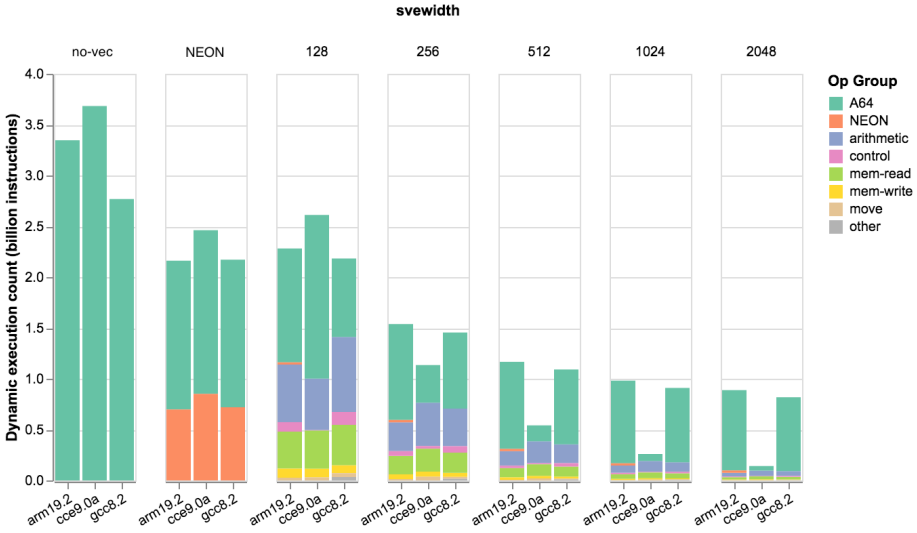

Figure 3. Dynamic instruction count and grouping for CloverLeaf.

MegaSweep was only vectorised by the Cray Compiler. As with STREAM, CCE performed control flow using scalar instructions, so the instruction counts followed a similar profile here. Because the GCC- and Arm-compiled versions were not vectorised, all instructions run were scalar A64 and their execution count did not change with SVE width.

Figure 4 shows the dynamic instruction analysis for **MiniFMM**. This application's build system does not currently support the Cray Compiler, so results are only shown for GCC and Arm. Even though the application was (partially) vectorised, the instruction count did not decrease significantly when increasing the SVE vector width over 512 bits, in contrast to the applications presented previously. Due to an interaction between the way MiniFMM vectorises over particles and the small scale of the problem run, not all the lanes in SVE registers were being utilised at high vector lengths; since the vectors were partially empty, the total instruction count did not decrease linearly.

Neutral is excluded from this analysis because it was not vectorised at all.

4.3 SVE Vector Lane Utilisation

Because SVE instructions employ per-lane predication, observing that SVE instructions are being *executed* is not enough to conclude that the application

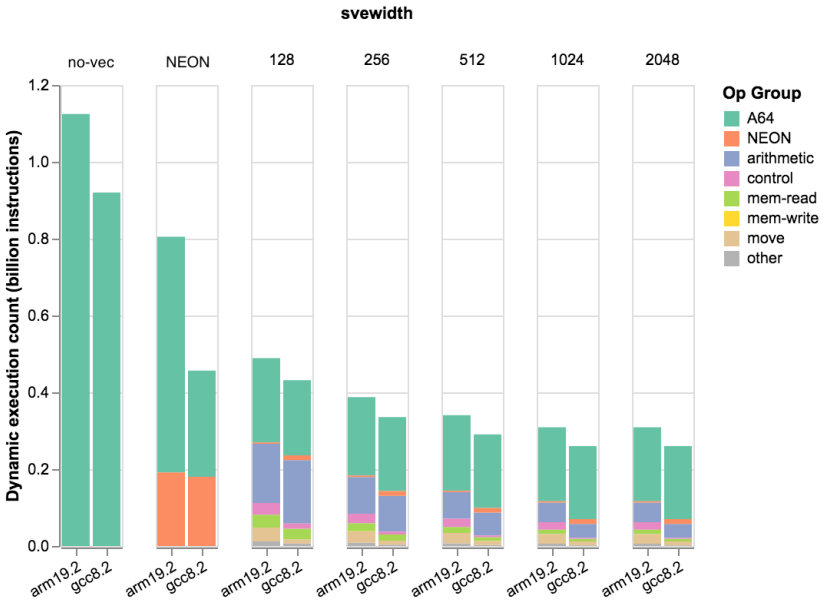

Figure 4. Dynamic instruction count and grouping for MiniFMM.

is using vector operations efficiently — it is possible that a large portion of the elements, potentially all but one, are masked out. This means that vector register can be underpopulated, almost empty. To investigate this, we looked at per-lane utilisation of SVE registers when running the mini-apps.

For applications with a high degree of vectorisation, e. g. BUDE, TeaLeaf, or CloverLeaf, vector operations were performed using all the lanes, i. e. at maximum utilisation. For MiniFMM, however, the number of active lanes varied: at 512-bit-wide SVE and below, most instructions used 80% or more of the lanes available, but when increasing the SVE length further, vector register utilisation peaked between 512 and 768 bits. Vector utilisation was virtually identical across both compilers tested, Arm and GCC.

Figure 5 shows a histogram of the number of active bits in SVE operations, grouped in 128-bit-wide bins. Increasing the SVE width past 512 bits brings little benefit for MiniFMM, as only a minority of the operations performed use more than 512 bits. When the vector width is set to 1024 bits, less than 5% of the instructions use the full available width, and further increasing the width to 2048 bits produces no change in vector utilisation.

In contrast, Figure 6 shows how BUDE, a mini-app that vectorises efficiently, was able to fully utilise vectors in *all* operations, even at the highest widths allowed by SVE. The other mini-apps investigated in this paper showed the same perfect vector utilisation efficiency as BUDE. These results cover both 32- and 64-bit floating-point data types: BUDE uses 32-bit data (`float`), and the other mini-apps use 64-bit types (`double`).

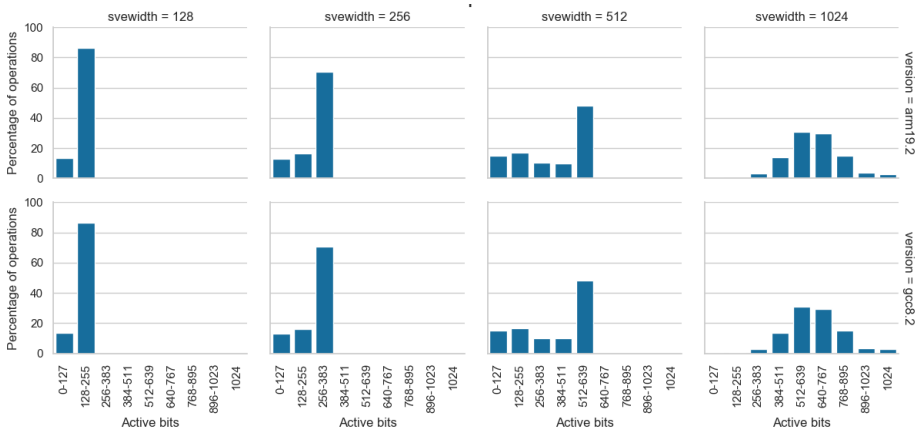

Figure 5. Histogram showing the number of active bits in the SVE operations performed by MiniFMM. The application cannot saturate the full widths of the vectors when the SVE length is 512 bits or higher.

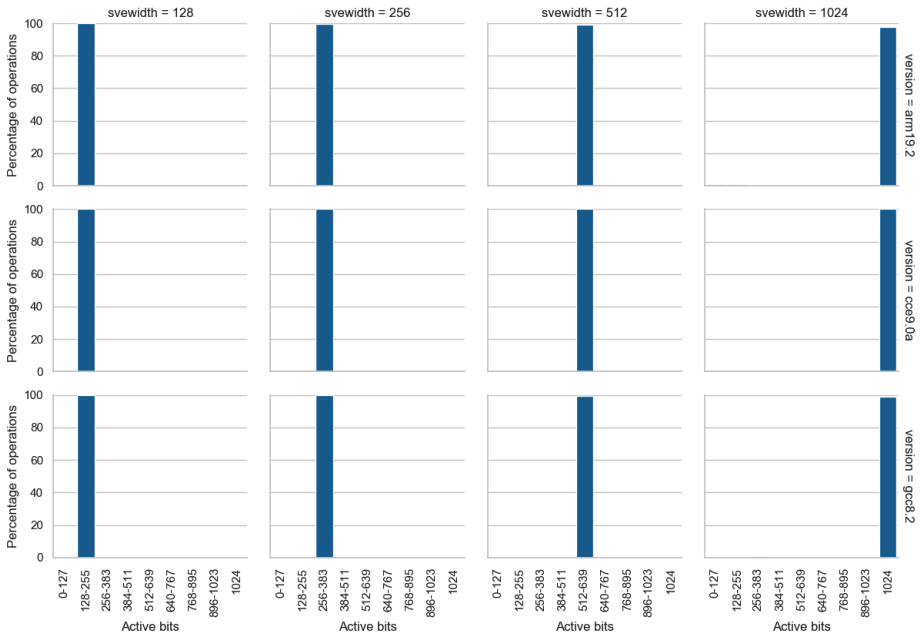

Figure 6. Histogram showing the number of active bits in the SVE operations performed by BUDE. Vectorisation is perfectly efficient at all SVE widths.

4.4 SVE Memory Operations

Finally, we looked at how the mini-apps are able to take advantage of SVE for memory operations. Since all SVE instructions are predicated per-lane, including contiguous and strided memory operations, every SVE memory instruction can differ in the number of bytes transferred.

We found that SVE usage for memory operations varied greatly between applications. Mini-apps with lower degrees of vectorisation, such as MegaSweep, used little SVE for memory accesses, but even applications with a higher degree of vectorisation showed a mixture of SVE and non-SVE memory operations. In BUDE, about three quarters of the memory instructions were SVE instructions; in CloverLeaf, TeaLeaf, and MiniFMM, between a quarter and a third of the memory operations were SVE. In MiniFMM, of the SVE operations, about a third were gathers, while there were no scatters; the other applications utilised contiguous accesses almost exclusively. All applications utilised all the SVE lanes in their memory operations, except for MiniFMM, where about half the SVE memory operations, including all the gathers, were only partially filled.

Figures 7 and 8 show the distributions of memory accesses in BUDE and MiniFMM, respectively. These two mini-apps form the most contrasting pair in the set of mini-apps evaluated. The observations here are consistent with Sections 4.2 and 4.3: BUDE vectorises very efficiently, and MiniFMM utilises some SVE-specific features but does not always utilise all vector lanes available.

These results are collected from the version of the applications compiled with the Arm Compiler 19.2 and run on 512-bit SVE, which is the vector length utilised in the upcoming Fujitsu A64FX processor. The absolute numbers of vector operations varies between the versions built with different compilers and when adjusting the SVE width, but the same important characteristics can be seen in all cases, and the conclusions drawn are similar.

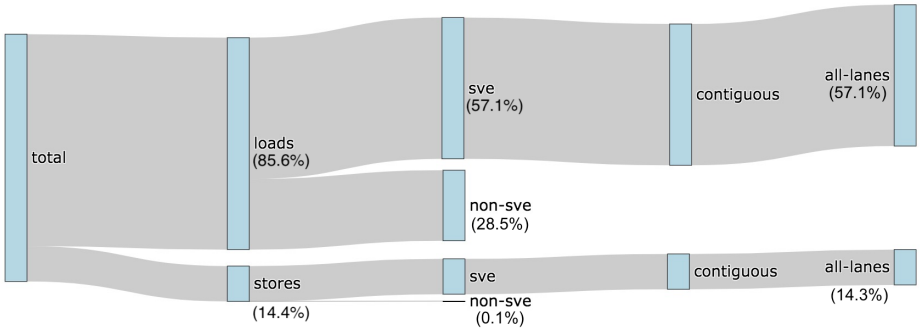

Figure 7. Relative counts, by number of instructions, of memory operations in BUDE. All memory accesses are contiguous and most are performed through SVE instructions.

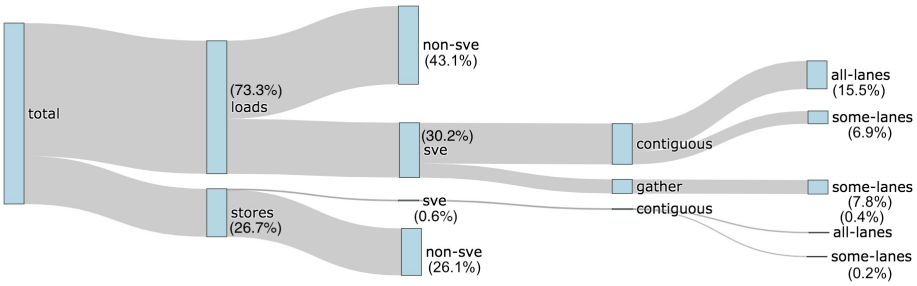

Figure 8. Relative counts, by number of instructions, of memory operations in MiniFMM. This applications shows a mixture of SVE and non-SVE operations, and the SVE ones show a further split between contiguous and non-contiguous accesses. Not all lanes are always used in SVE operations for MiniFMM.

5 SVE Usage Discussion

The **STREAM** benchmark runs simple, predictable memory operations. All the compilers tested were able to successfully use SVE — at all vector lengths — to vectorise this code, and at run-time the vectors were fully utilised. This is the expected behaviour for the benchmark.

BUDE is a heavily compute-bound benchmark, and thus complementary to STREAM. This application shows very efficient utilisation of SVE: the main kernels all execute vectorised operations, which scale with the chosen SVE length. At 128 bits, the amount of code run — both vector and scalar — is almost identical to the established NEON version, which indicates that good code is generated by all the compilers. Increasing the vector length by 2× reduces by half the number of instructions run up to 1024 bits; at 2048 bits, the total number of executed vector instructions becomes smaller than the number of scalar instructions.

Even though more than half of the main loops in **TeaLeaf** are vectorised by all the compilers, only relatively few vector instructions are executed at runtime: for 128-bit SVE, these represent less than a third of the total instructions run for the Arm and GCC versions. Increasing the vector length decreases the count, but only with around 50% efficiency and up to 1024 bits; there is virtually no change going to 2048 bits. The Arm-compiled executable runs comparatively more instructions than the GCC version, by 35–40%, depending on the chosen vector length. With the Cray executable, less than 10% of the instructions run are vector operations, even though the compiler vectorised the same loops as Arm and GCC; at 1024 and 2048 bits, the vector code run is NEON, and not SVE, which we suspect is due to a compiler bug.

The **CloverLeaf** benchmark shows characteristics similar to TeaLeaf, but with more vector instruction utilisation. In all three versions, vector instructions account for between a third and half of the total instruction count at 128 bits; all three compilers produce a similar total dynamic instruction count. The SVE instruction count scales as expected up to the largest vector width possible, 2048 bits. The Cray-compiled version initially runs the highest number of total

instructions, but it decreases sharply at 256 and 512 bits; at 512 bits more than two thirds of the code executed is SVE, and at 2048 bits the total count constitutes 22% of those of the Arm and GCC versions, suggesting that the Cray compiler optimises better for higher vector lengths.

This also hints at the importance of the loop chosen for vectorisation: if a compiler is able to vectorise the outer loop, as CCE is, and perhaps also to collapse the inner loop when doing so, the reduction in instruction count at high vector lengths can be considerable. On the other hand, the same strategy may not be desirable at smaller vector lengths, where vectorising the inner-most loop may be optimal. This would imply that, for optimal code generation, the compiler either needs to know the hardware vector width at compile-time, or it needs to generate several code paths and dynamically choose the optimal one when the vector length information becomes available at run-time.

A related issue is that the compilers tested in the study use a generic cost model model for SVE, which may not accurately reflect any real implementation. With access to the cost model of a real SVE processor, the compilers may generate different code to take advantage of the implementation's strengths.

In CloverLeaf, SVE memory accesses represent about half the total memory operations performed, both when reading and writing, and the vast majority of those are contiguous operations.

Of the mini-apps included in this study, **MegaSweep** shows the most notable difference between the three compilers: Cray is the only one that successfully vectorises the code, both on NEON and SVE. The binary it produces runs 2.5× fewer total instructions than Arm and GCC at 128 bits, and the amount of SVE instructions executed scales almost perfectly up to 2048 bits, although the 1024-bit binary highlights a compiler issue where some of the code run is NEON, not SVE, which reduces the scaling efficiency in this particular case. At 2048 bits, the Cray version runs 10.5× fewer instructions than the GCC alternative. The Cray version also successfully utilises SVE for memory access, all of which are contiguous and are able to exploit the full lengths of the vectors.

Neutral does not vectorise with any of the compilers, so no SVE is being run. Due to the nature of the Monte-Carlo algorithm, there is little structure in the access patterns in the kernels. As Martineau and McIntosh-Smith explained, it is possible to force vector code generation, but it will be comprised almost entirely of indirect, variable-stride accesses that do not improve performance [21]; the compilers make the right choice to generate scalar instructions in this case.

In general it is desirable to utilise as much of the available vectors as possible, but partial utilisation does not always signal a problem. The **MiniFMM** result exhibits the flexibility of SVE: even though the parallelisation strategy in the application cannot fill the vectors above 512 bits, the hardware can still efficiently utilise its resources by executing partially masked operations. These operations should not be any more expensive than regular operations with full vectors, and so are more efficient than falling back to scalar code.

6 Relevance of SVE for HPC

The results presented in Sections 4 and 5 show that SVE is a viable, competitive vector instruction set for HPC applications. For HPC workloads, it represents a noticeable improvement over NEON, bringing high-performance Arm processors in line with current-generation x86 processors, both in terms of the available vector length and the flexibility of the operations.

Even though no SVE hardware is currently available, we have found the SVE toolchains to be mature already. Generating SVE code only required enabling the SVE extension in the target architecture flag, and the compilers were successful in utilising SVE where expected. Compared to NEON, more loops were vectorised with SVE by all compilers, and the Arm and Cray compilers achieved a similar or higher degree of vectorisation on SVE compared to AVX-512.

One of the main advantages of SVE arose from its per-lane predication, which allowed loops with heavy control flow to be vectorised without additional cost. This additional flexibility meant it was sometimes beneficial to vectorise loops on SVE even when it was not on other instructions sets.

In the wider context, these results suggest that many HPC applications should be able to utilise SVE and benefit from doing so. The flexibility of SVE allows a wide range of loops to be turned into vector code, including cases where vectorisation is not possible with NEON or AVX, e. g. with irregular and unpredictable access patterns. Compute-bound applications can exploit high vector widths, bringing the number of instructions required significantly lower than on (128-bit) NEON. Partially filled operations allow vector instructions to be generated and executed even when the application cannot fill whole vector registers, a more efficient alternative than falling back to scalar code.

While in this study we have shown that SVE HPC applications behave well in an *emulated* environment, we cannot make any claims regarding their performance on real hardware. Implementations of SVE are likely to come with caveats and performance characteristics which cannot be determined *a priori*, and so it is impossible to predict which types of operations will be fast and which will bring little improvement over scalar code. There are currently no widely available tools to generate and run SVE code tuned for a specific microarchitecture definition, without which such a study is infeasible.

7 Future Work

The analysis presented in this study covers the three main SVE compilers available at the time of writing. However, Fujitsu A64FX systems are expected to be available in the near future, and Fujitsu will supply a proprietary compiler to accompany their processor. Optimisations applied by this compiler may be key in extracting high performance from the A64FX, so analysing the binaries it produces should prove a valuable research direction.

Further work will be enabled when the compilers are able to generate *tuned* binaries. The early versions used in this study only use a generic model of an SVE processor, because neither the compilers nor ArmIE currently allow the user to specify microarchitectural details, except the SVE width. Once a tuned

binary can be generated, running it on its target platform will enable quantifying of the tuning benefit, and an even wider range of experiments is possible if these tuning parameters can be adjusted dynamically. This class of experiments for microarchitectural design-space exploration with arbitrary hypothetical processor configurations is one of the main goals of the upcoming SimEng simulator developed at the University of Bristol [23].

Finally, evaluation of full-size HPC applications on real inputs is intractable with the currently available emulation tools. The overhead incurred by ArmIE increases by several orders of magnitude when the instrumented application needs to use system calls, dynamically linked libraries, and file operations. For such experiments, benchmarking real hardware remains the only viable option.

8 Reproducibility

All mini-apps used in this study are open-source software and can be downloaded from their respective homepages. Detailed build and run instructions for each application, the custom ArmIE instrumentation clients used for this paper, and scripts to aggregate and plot the collected data can be found at `https://github.com/UoB-HPC/sve-analysis-tools/tree/euro-par-2020`.

9 Conclusion

In this work, we have presented an analysis of SVE usage across a number of mini-apps that span several common HPC problem classes. We have looked at how currently available compilers are able to utilise SVE to automatically vectorise the mini-apps' code, how much of the executed code is SVE, the efficiency of the executed SVE vector instructions, and whether new ways of accessing memory introduced with SVE are utilised in these mini-apps.

We found that SVE was generally well targetted by the compilers: in most cases, compilers were able to utilise SVE at least as well as AVX and NEON, and often better. The available compilers for SVE were only surpassed by the Intel compiler targetting AVX on select few occasions. Most SVE binaries used wide vectors efficiently, with all lanes being active for the vast majority of the run time; MiniFMM was the only exception, where SVE efficiency varied depending on the SVE width utilised. In terms of memory accesses, vectorised mini-apps were able to use SVE instructions to efficiently load and store data, and MiniFMM also made use of gather operations, either fully or only partially filled. We saw little use of SVE scatter instructions, but this is expected given the optimised memory access patterns on the mini-apps studied.

We conclude that SVE is a promising instruction set, and HPC applications and toolchains appear ready to take advantage of it to deliver performant code running on upcoming generation of Arm-based high-performance processors.

References

[1] Simon McIntosh-Smith, James Price, Tom Deakin and Andrei Poenaru. 'A performance analysis of the first generation of HPC-optimized Arm processors'. In: *Concurrency and Computation: Practice and Experience* 31.16 (2019), e5110. DOI: 10.1002/cpe.5110.

[2] Nigel Stephens et al. 'The ARM scalable vector extension'. In: *IEEE Micro* 37.2 (2017), pages 26–39. DOI: 10.1109/MM.2017.35.

[3] Tiffany Trader. *Cray, Fujitsu Both Bringing Fujitsu A64FX-based Supercomputers to Market in 2020*. HPC Wire. 12 Nov. 2019. URL: https://www.hpcwire.com/2019/11/12/cray (visited on 10/12/2019).

[4] T. Deakin, S. McIntosh-Smith and W. Gaudin. 'Expressing Parallelism on Many-Core for Deterministic Discrete Ordinates Transport'. In: *2015 IEEE International Conference on Cluster Computing*. Sept. 2015, pages 729–737. DOI: 10.1109/CLUSTER.2015.127.

[5] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy and S. A. Jarvis. 'Exploring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors'. In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. May 2013, pages 1085–1097. DOI: 10.1109/IPDPS.2013.44.

[6] S. Hammond, C. Vaughan and C. Hughes. 'Evaluating the Intel Skylake Xeon Processor for HPC Workloads'. In: *2018 International Conference on High Performance Computing Simulation (HPCS)*. July 2018, pages 342–349. DOI: 10.1109/HPCS.2018.00064.

[7] Saeed Maleki, Yaoqing Gao, Maria J. Garzar'n, Tommy Wong and David A. Padua. 'An Evaluation of Vectorizing Compilers'. In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. Galveston, TX, USA: IEEE, Oct. 2011, pages 372–382. ISBN: 978-1-4577-1794-9. DOI: 10.1109/PACT.2011.68.

[8] Angela Pohl, Biagio Cosenza and Ben Juurlink. 'Portable cost modeling for auto-vectorizers'. In: *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2019, pages 359–369.

[9] Bo Zhao et al. 'Performance Evaluation of NPB and SPEC CPU2006 on Various SIMD Extensions'. In: *Big Data Computing and Communications*. Springer International Publishing, 2015, pages 257–272. ISBN: 978-3-319-22047-5.

[10] Lynd Stringer. *Vectors: How the Old Became New Again in Supercomputing*. HPC Wire. 26 Sept. 2016. URL: https://www.hpcwire.com/2016/09/26/vectors (visited on 11/12/2019).

[11] Paul Stewart Crozier et al. *Improving performance via mini-applications*. Technical report SAND2009-5574. 1 Sept. 2009. DOI: 10.2172/993908.

[12] Simon McIntosh-Smith, James Price, Andrei Poenaru and Tom Deakin. 'Benchmarking the first generation of production quality Arm-based supercomputers'. In: *Concurrency and Computation: Practice and Experience* (2019), e5569. DOI: 10.1002/cpe.5569.

[13] Toshio Yoshida. 'Fujitsu high performance cpu for the post-k computer'. In: *Hot Chips 30 Symposium (HCS), Series Hot Chips*. Volume 18. 2018.

[14] David Schor. *Marvell Lays Out ARM Server Roadmap*. WikiChip Fuse. 9 Nov. 2019. URL: https://fuse.wikichip.org/news/2956/marvell-lays-out-arm-server-roadmap (visited on 10/12/2019).

[15] Yuetsu Kodama et al. 'Preliminary Performance Evaluation of Application Kernels Using ARM SVE with Multiple Vector Lengths'. In: *2017 IEEE International Conference on Cluster Computing*. IEEE. 2017, pages 677–684.

[16] John D McCalpin. 'Memory bandwidth and machine balance in current high performance computers'. In: *IEEE computer society technical committee on computer architecture (TCCA) newsletter* 2.19–25 (1995).

[17] Simon McIntosh-Smith, James Price, Richard B Sessions and Amaurys A Ibarra. 'High performance in silico virtual drug screening on many-core processors'. In: *The International Journal of High Performance Computing Applications* 29.2 (2015), pages 119–134. DOI: 10.1177/1094342014528252.

[18] Matthew Martineau, Simon McIntosh-Smith and Wayne Gaudin. 'Assessing the performance portability of modern parallel programming models using TeaLeaf'. In: *Concurrency and Computation: Practice and Experience* 29.15 (2017). ISSN: 15320626. DOI: 10.1002/cpe.4117.

[19] A.C. Mallinson et al. 'CloverLeaf: Preparing Hydrodynamics Codes for Exascale'. In: *Cray User Group*. Napa Valley, California, USA, May 2013.

[20] Tom Deakin, Wayne Gaudin and Simon McIntosh-Smith. 'On the Mitigation of Cache Hostile Memory Access Patterns on Many-Core CPU Architectures'. In: *High Performance Computing*. Springer International Publishing, 2017, pages 348–362. ISBN: 978-3-319-67630-2.

[21] M. Martineau and S. McIntosh-Smith. 'Exploring On-Node Parallelism with Neutral, a Monte Carlo Neutral Particle Transport Mini-App'. In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2017, pages 498–508. DOI: 10.1109/CLUSTER.2017.83.

[22] P. Atkinson and S. McIntosh-Smith. 'On the Performance of Parallel Tasking Runtimes for an Irregular Fast Multipole Method Application'. In: *Scaling OpenMP for Exascale Performance and Portability*. Springer International Publishing, 2017, pages 92–106. ISBN: 978-3-319-65578-9.

[23] Simon McIntosh-Smith. 'Enabling Processor Design Space Exploration with SimEng'. In: ModSim: Workshop on Modeling and Simulation of Systems and Applications (2019). Seattle, WA, Aug. 2019.